

An Empirical Analysis of Techniques for Constructing and Searching K-Dimensional Trees

Douglas A. Talbert and Doug Fisher

Presented By:

Manish Pandya

Hemin Desai

Chirag Shah

Summary

Affordable, fast computers with large memories have lessened the demand for program efficiency, but applications such as browsing and searching very large databases often have rate limiting constraints and therefore benefit greatly from improvements in efficiency. This paper empirically evaluates several variants of a common k-dimensional tree technique to demonstrate how different algorithm options influence search cost for nearest neighbors.

Introduction

The memory and speed of today's computers seem to obviate the need for the peak efficiency that minimal memory and relatively slow speeds used to demand. However, there remain computing scenarios with significant rate-limiting components in which slight efficiency improvements provide significant performance gains.

A common organizational structure used for nearest-neighbor searching is a k-dimensional tree (kd-tree). Kd-trees are designed for efficient nearest-neighbor search. This paper presents an empirical analysis of kd-tree nearest-neighbor searches with respect to several independent variables. Author's goal is to demonstrate the effect on search cost of various kd-tree construction and search options and to explain the exhibited relationships to enable researchers or automated agents to adapt a kd-tree algorithm to better address their needs.

Background

Kd-Trees

Kd-trees are multi-dimensional binary search trees that organize vectors of numeric data to facilitate nearest-neighbor searches. Each internal node of the tree represents a branching decision in terms of a single attribute's value, called a split value. All data with values less than or equal to the split value along that dimension follow one branch and all data with values greater than the split value along that dimension follow the other branch. Each leaf is a collection of data items. The number of items in each leaf is bounded by a user-specified threshold.

Search Algorithm

Given a kd-tree constructed over a set of numeric vectors, D , the search for the nearest neighbor of a numeric vector, v , in D proceeds as follows:

1. Starting at the root, use the values in v to follow the branches of the kd-tree to an initial leaf.
2. Compute the distance to each data item at the leaf; store the closest item and its distance.
3. Continue searching the tree in a depth-first fashion, using the values in v to determine which branch to search first at each internal node.
4. Whenever a leaf is encountered, compute the distances to each data item at the leaf and update the nearest-neighbor information when needed.

This algorithm does not address two values raised by missing values.

1. how to decide which branch to follow when v has no value for the node's split attribute
2. how to compute a distance when either v or a data item from D is missing one or more values.

When v is missing a value for a split attribute, we pick the "less than or equal to" branch first and explore the "greater than" branch later in depth-first search.

Attributes for which v has no value are considered irrelevant and can be ignored in distance calculations. If, however, v has a value for an attribute and a data item does not, we need to somehow include that attribute in the calculations.

We assume that the missing value is as distant from the known value in v as possible given the known range for that attribute. Assuming that the range exhibited for each attribute in the data set is accurate, this approach guarantees that the data item identified as the nearest neighbor is correct even if that item has missing values for relevant attributes.

The algorithm above assumes interest in the nearest single neighbor. In general, one can search for as many “nearest” neighbors as desired.

Search Cost

Since we are interested in measuring search efficiency we must quantify search cost. Search cost includes search time, number of nodes examined during search, number of distance calculations performed and the number of dimensional comparisons performed during search. (A dimensional comparison occurs once at each internal node to decide which branch to search first and many times at each leaf during distance calculation.)

Authors have used number of nodes and number of dimensional comparisons to evaluate kd-trees. The number of dimensional comparisons is a true measure of cost under ideal circumstances, but the number of nodes may better reflect disk access when data must be stored on secondary memory.

Reducing Search Cost

The structure of kd-trees can be exploited to improve the search algorithm. Because each split in the kd-tree is based on a single value of a single attribute, each node in the tree covers a hyperrectangular region of the search space. The root covers the entire search space, and the space covered by any other node in the tree is bounded by the hyperplanes defined by the split points in all its ancestor nodes.

To exploit this property, we must recognize that the search space in which a possible nearest neighbor might be found is always a hypersphere. The radius of this hypersphere is the distance to the current nearest neighbor. If multiple nearest neighbors are sought, the radius of the hypersphere is the distance to the farthest item in the set of candidate nearest neighbors. The relationship between the hypersphere of relevant search space and the hyperrectangle covered by a node can be used to prevent unnecessary searching.

Strong Pruning

This pruning technique checks to see whether a node needs to be examined. If a node's hyperrectangle does not intersect the hypersphere of relevant search space, then neither that node nor the subtree beneath it can contain any data items closer than the current candidate nearest neighbor(s).

This is strong pruning because it only allows searching a node if the hyperrectangle and the hypersphere do indeed overlap. It is performed prior to searching the second child of any node.

Weak Pruning

Strong pruning check is relatively expensive and that it may be approximated by checking if the hypersphere of relevant search space intersects the hyperplane formed by the node's split point.

When the strong pruning check is true, then the weak check is also true, but the converse is not true. The weak check is less costly to perform but results in more nodes being searched. This is because it does not examine all the bounds on a hyperrectangle; it only examines one hyperplane boundary. Bounds from other ancestor nodes might indicate that there is no need to search that space, but this check can't know that.

Stopping Search Early

If the hyperrectangle of a node that has been completely searched contains the entire hypersphere of relevant search space, then the search can stop because we are guaranteed to have the correct nearest neighbor(s). It is checked prior to backtracking out of each node.

Reduced Distance Calculations

If the data vectors are defined using 50 attributes, then normally each distance calculation would require 50 dimensional computations. If, however, after the first two dimensional computations, the distance is already greater than the distance to the current candidate nearest neighbor, no more useful calculations need to be performed. Thus, 48 dimensional computations would be eliminated.

Construction Algorithm

The basic kd-tree construction algorithm can be described as follows:

1. Select a split point.
2. Partition the data accordingly.
3. Repeat until the size of each node is less than or equal to a user-specified threshold. These nodes are the leaves of the kd-tree.

You can split on the median of the attribute with the broadest range at each node. The arithmetic mean provided a better split point because it results in hyperrectangle dimensions with more uniform sizes.

In addition to a split-point algorithm, the maximum size of a leaf must be chosen prior to tree construction. Techniques need to be in place for handling missing values during tree construction and for normalizing the data prior to tree construction. If an element has no value along the split-point attribute, always place that item down the “less than or equal to” branch. The search algorithm guarantees that if that item is the nearest neighbor, it will be found.

If the data’s attribute uses different scales of measurement, it may be necessary to normalize the data because those differences could skew the distance calculations.

Algorithms

Kd-tree construction has at least two elements that can vary.

1. the split point selection algorithm
2. the maximum number of data items in a leaf

Kd-tree search has at least three variables.

1. the pruning technique that will be used
2. whether or not to check for early completion of the search
3. whether or not to perform reduced distance calculations

The choices made on these options influence the cost of the search.

For tree construction, the tested leaf sizes are 1, 2, 5, 10, 20 and 50 and the split point algorithms they tried were median, arithmetic mean, harmonic mean and inner quartile mean of the attribute with the broadest range at each node.

For searching, the pruning techniques they evaluate include no pruning, strong pruning and weak pruning. They also consider a hybrid pruning method that relies on weak pruning as long as it indicates not to search a node but strong pruning verify that the other nodes do indeed need to be searched. Furthermore, they evaluate the utility of checking for early search completion and of performing reduced distance calculations.

Example

A kd-tree might contain information on all cities with post offices. Associated with each city is its longitude and latitude. If a letter is addressed to a town without a post office, the closest town that has a post office might be chosen as the destination. This town can be found by nearest neighbor search from a kd-tree.

Results

Split Point Selection Algorithm

Of the split point algorithms examined, the mean enables the most efficient search across a variety of pruning methods and maximum leaf sizes. As leaf size increases, cost in nodes decreases. The weak pruning is more sensitive to inefficiencies in the tree because it prunes less than strong pruning.

Leaf Size

Average cost in nodes decreases with increasing leaf size. The strong pruning minimizes cost at an intermediate leaf size while weak pruning monotonically increases with leaf size.

Searching with strong pruning has two different primary costs. It has the expensive distance calculations at the leaves that all kd-tree searches have, but it also has an expensive pruning check. When a tree has many very small leaves, the pruning check has to be performed very frequently driving cost up, and when the leaves are significantly larger, the pruning can't provide a fine-grained search focusing on only the most relevant parts of the search space. Between these extremes, there is optimal leaf size at which the pruning checks are not excessive and the leaves cover adequately small portions of the search space.

Weak pruning's monotonic increase is because, unlike the strong technique, its pruning check is inexpensive. Thus its relative cost across leaf sizes is influenced primarily by how precise the pruning can be. Larger leaves result in less pruning precision and higher costs.

Strong vs. Weak Pruning

They also tested a hybrid pruning approach in which strong pruning was only used when weak pruning indicated that a node needed to be searched. This guarantees the same number of nodes will be searched as when strong pruning

is used by itself but should result in a lower cost in dimensions because some nodes will be pruned using the weak method alone.

The number of nodes searched using the hybrid approach was indeed the same as for strong pruning. Hybrid pruning is a modest improvement over strong pruning in terms of the number of dimensions.

Another measure of how well a pruning technique works can be seen in amount of “wasted” search. In kd-trees that would be a measure of how much search had to occur after finding the nearest neighbor.

Reduced Distance Calculations

Costs were lowered for reduced distance calculations. The lower pruning precision caused by large leaves is less costly.

Conclusion

In terms of dimensions examined, the weak pruning method seems advantageous, for small leaf sizes.

In contrast, strong pruning appears to be preferred in terms of the number of nodes visited, particularly at large leaf sizes.

Reduced distance calculations is better no matter what pruning method is used. Large amount of search is spent after the nearest neighbor(s) has been found.